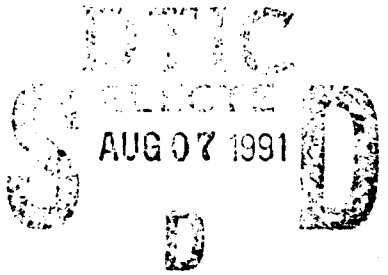
(2)

Technical Report 1429
June 1991



Software Development on the High-Speed Systolic Array Processor (HISSAP):

Lessons Learned

F. M. Tirpak, Jr.

NAVAL OCEAN SYSTEMS CENTER
San Diego, California 92152-5000

J. D. FONTANA, CAPT, USN
Commander

R. T. SHEARER, Acting
Technical Director

ADMINISTRATIVE INFORMATION

The work detailed in this report was performed by the Processing Research and Development Branch (Code 761) of the Naval Ocean Systems Center, San Diego, CA 92152-5000. The project number is EE34, agency accession number DN308022, and program element NIF, 604507N.

Released by
G. W. Byram, Head
Processing Research and
Development Branch

Under authority of
F. M. Tirpak, Sr., Head
Space Systems and
Technology Division

ACKNOWLEDGMENT

The High-Speed Systolic Array Processor (HiSSAP) project encompasses work performed over a 7-year period at NOSC. The goal of this project was to obtain a clearer understanding of the complex interactions between parallel processing architectures and adaptive matrix-based signal-processing algorithms. Sponsorship of system hardware and software development was initially obtained from the Lasers and Microelectronics NOSC program block managed by Dr. Isaac Lagnado, and the NOSC Research and Technology Branch (Code 553). During the intermediate phase of the project, mapping of the multiple-signal classification (MUSIC) algorithm and the finite-impulse response (FIR) filter onto the testbed hardware was supported by joint sponsorship of the block and major bid and proposal discretionary funding (coordinated by Dr. John Silva, formerly NOSC Program Director for Research and Technology). Integration of these system software applications, and the balance of the signal processing/data acquisition, and the demonstration of the high-frequency, direction-finding application was obtained from the NAVSEA standard matrix processor project (PMS-412), with additional funds supplied by the Office of Naval Technology's Lasers and Microelectronics block.

SUMMARY

OBJECTIVE

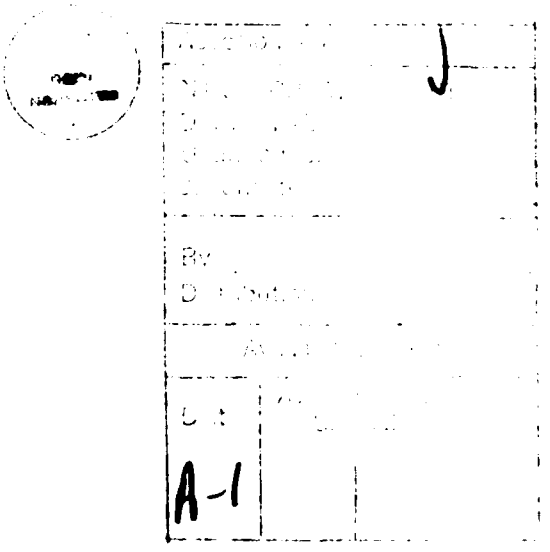
This report documents the "lessons learned" in programming the Naval Ocean System Center's (NOSC's) High-Speed Systolic Array Processor (HISSAP) testbed. The procedures used for code generation, along with the programming utilities provided in the software development environment, are discussed with regard to their impact on the efficient implementation of algorithms on a parallel processing system such as HISSAP. This information is intended for considerations pertaining to software-development environments in future Navy parallel processing systems.

RESULTS

Many of HISSAP's software-development utilities played key roles in the implementation of two computationally intensive algorithms: the Multiple-Signal Classification algorithm (MUSIC) and a four-channel, narrowband, finite-impulse response (FIR) filter. The introduction of utilities not included with the HISSAP tools would undoubtedly have increased the speed and efficiency of software development.

RECOMMENDATIONS

Commercial software development environments (targeted for processing platforms where large volumes of application software are to be output) generally include features more advanced than those of HISSAP. However, to meet minimum requirements for the efficient implementation of algorithms, some utilities are indispensable. Some of these basic software development utilities, whether present with the HISSAP testbed or not, are discussed in this report.



CONTENTS

FIGURES

INTRODUCTION

This report documents the software integration procedure on the Naval Ocean Systems Center's (NOSC's) High-Speed Systolic Array Processor (HISSAP) testbed. The report draws from the experience gained in implementing application-specific software. The procedure is evaluated to provide some "lessons learned" as they apply to algorithm development on future Navy signal-processing systems.

BACKGROUND

The Navy is planning to extend the AN/UYS-2(V) Enhanced Modular Signal Processor (EMSP) by providing a new functional element type, the Matrix Processor (MP). This new element will be a parallel processing array optimized for the efficient implementation of linear algebra operations. Planning and specification of the MP have made extensive use of the existing Navy in-house experience base in systolic systems. This experience includes an ongoing investigation at NOSC of the applicability of parallel processing systems to certain signal-processing tasks. Processing systems developed during this time, starting in 1979, include the Systolic Array Processor (SAP), the Systolic Linear Algebra Parallel Processor (SLAPP), the Video Analysis Transputer Array (VATA), and the High-Speed Systolic Array Processor (HISSAP) testbed. Of these systems, algorithm mapping work continues on the VATA; in the near future, mapping work will begin on an Intel iWarp* parallel processing array. Further information on these systems is given in references 1, 2, 3, and 4.

HISSAP, the system documented in this report, was built to host several signal-processing algorithms including a multiple-channel narrowband filter and the Multiple-Signal Classification (MUSIC) algorithm. The HISSAP study concluded with a successful laboratory test of a high-frequency direction-finding (HFDF) application. As a short summary, the test setup consisted of two "HF signals" (two laboratory synthesizers operating in the low HF band) transmitted through an analog antenna array simulator, digitized by a data-acquisition subsystem, and processed by HISSAP, which computed the simulated directions of arrival. HISSAP processed the digitized data by using the two signal-processing routines mentioned above (the digital filter and MUSIC). The results of this test are documented in reference 5.

SCOPE

The HISSAP HFDF effort provided insight into the complex interdependencies between architectures and algorithms. In addition, because the HFDF application was

*INTEL and iWarp are trademarks of Intel Corporation.

the first of the NOSC-developed systems to feature a complex software development environment, the application was an important source of in-house experience in developing and integrating parallel-processing software development tools.

Software development is a critical aspect of any parallel processing system; in general, as high-performance processing architectures and applications become more complex, the software development process becomes increasingly difficult. As a result, application software takes longer to produce. Programmers require powerful tools to help reduce the lag time between problem identification and algorithm implementation.

The original goal of the HISSAP project involved the study of parallel processor/parallel algorithm interactions. A natural extension of this study would be to evaluate and improve parallel programming environments. In light of the recent increase in commercial activity involving software development environments, this report provides guidance, through NOSC's first-hand experience, to assist in developing future programming environments for Navy systems.

HISSAP PROGRAMMING PROCEDURE: AN OVERVIEW

First, the report provides a short summary of the algorithm programming procedure on HISSAP. Next is a brief overview of the programming resources available in the software development environment. Finally, a commentary is made on parallelizing compilers and their applicability to the HISSAP effort.

SUMMARY OF PROGRAMMING PROCEDURE AND TOOLS

Figure 1 is a summary of the software development procedure. The process of programming an algorithm on HISSAP began with the conversion of a sequential model into parallel form, with the workload partitioned among multiple processors according to a predefined mapping strategy. The mapping strategy was usually the result of algorithm analysis that used the software package PC-MATLAB* (described later). Binary HISSAP native code (microcode) was then generated to control each processor's activity in the algorithm execution. The microcode files were downloaded to the appropriate processors for subsequent execution in a custom operating environment. An IBM PC/AT** served as the host computer to the HISSAP array, as well as the platform upon which software development took place.

*PC-MATLAB is a trademark of The MathWorks, Inc.
**IBM PC/AT is a trademark of IBM.

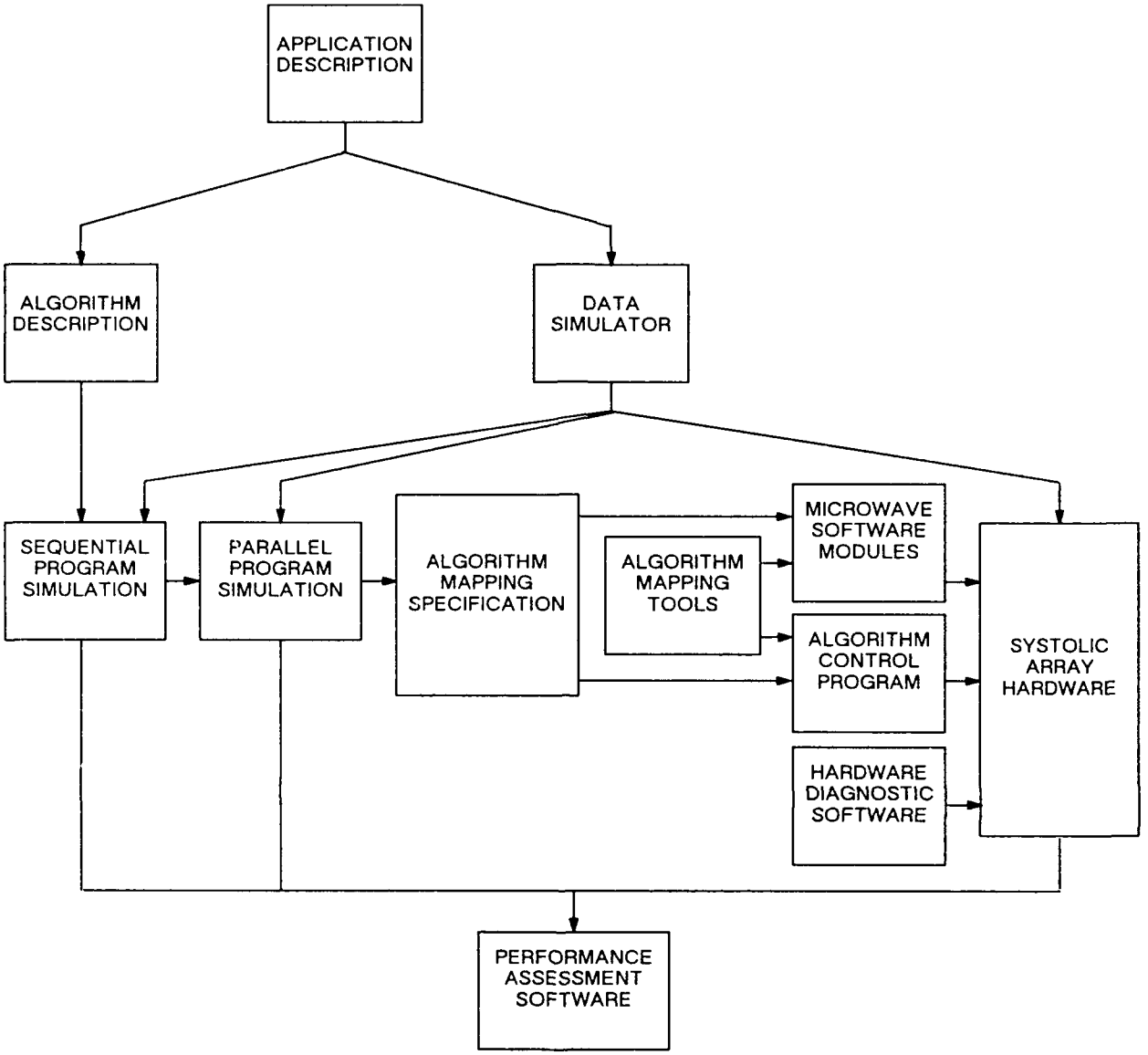


Figure 1. HISSAP software development procedure.

The HISSAP software development environment consisted of microcode generation utilities, subroutine object code libraries, software execution utilities, and run-time analysis and debugging tools. Each component made an important contribution to the software development phase of the HISSAP effort. These components, as well as the use of PC-MATLAB as an algorithm development tool, will be discussed in greater depth.

A WORD ABOUT COMPILERS

State-of-the-art, commercially available parallel processing systems employ *parallelizing compilers* in their programming environments. The HISSAP programming

environment does not. These powerful resources assist in the partitioning and mapping (*task allocation*) described above by customizing the algorithm to the system's underlying processor topology. *Resource allocation* (e.g., memory and floating-point register assignment) and *interprocessor communication* are accomplished similarly by these compilers, thereby greatly reducing the workload placed upon programmers. The study of algorithm-architecture interaction that uses HISSAP as a research platform has not included the development of a parallel compiler customized to HISSAP's unique processing characteristics.

It is expected that the Matrix Processor enhancement to the EMSP will include a parallelizing compiler resource. In fact, programming parallel algorithms at the level envisioned for Navy signal-processing applications would be impossible without such a resource. Therefore, as a first "lesson" learned by programming HISSAP, a massively parallel programming environment must have a parallelizing compiler to expedite code generation at the system level. This compiler would be customized to the system's architecture and thereby would be responsible for the machine-level issues involved in programming.

In many cases, the first attempts to program a sequential algorithm onto a parallel processing system are performed strictly "by hand," i.e., the allocation of tasks onto the processing elements is determined by the programmer and not by an "intelligent" compiler. Such was the case in programming MUSIC and the filter on HISSAP. However, as more and more routines become established in software, their implementations on such a system become simpler. The programmer identifies the routine and assigns a problem size, and the compiler, if available, partitions the problem and generates code.

Some of the topics discussed in the following sections relate to code development tasks normally accomplished by parallelizing compilers. However, even with the incorporation of such a resource in the HISSAP programming environment, most of the "lessons learned" referenced in the following sections DO NOT become trivial. The circumstances of the effort must be understood in order to take the topics discussed in the following sections in the proper vein. Since no straightforward method of programming the HFDF algorithms existed prior to their implementation on HISSAP, it is unclear what benefit, if any, a compiler would have provided. Therefore, so as not to belabor the obvious, the documentation will avoid the tendency to remedy problems with a compiler as a "cure-all," although it should be understood that a compiler is an essential part of any proposed programming resources.

PRIMITIVE PROGRAMMING

This section begins with a concise description of the primitive-level coding procedure on HISSAP. In this context, *primitive coding* refers to generating microcode that

describes a basic machine operation, or several such operations that comprise a basic algorithm subroutine. Primitive coding procedures are also important in other applications including "hard-wiring" array processor functions into other operations and writing optimized code. For example, the pipelining of arithmetic instructions is accomplished this way. The resulting primitive object code is stored in libraries for incorporation into higher level applications.

This section details some important primitive-level code generation utilities (both present in and absent from HISSAP). The section also describes system details upon which application-level programming utilities (discussed later) are based.

HISSAP CODE GENERATION UTILITIES

A HISSAP microword was constructed of 176 bits for the arithmetic processors (64 for the input/output processors), organized into single-bit or multiple-bit instruction fields. The fields controlled the various hardware subsystems such that their operations took place concurrently during an instruction cycle. In simplest form, programming HISSAP involved configuring the fields necessary to enable desired hardware functions during a given microword's execution. Each field had to be programmed in each instruction clock cycle. Moreover, some hardware functions required several clock cycles (hardware pipelining), so the related bit fields also had to be programmed correctly over multiple instructions. Generating and managing instructions of this complexity necessitated a sophisticated set of microcode development tools.

Extended C Functions

HISSAP microcode was created in one of two ways. In one method, generating primitives was accomplished while using custom functions written in C. Many basic machine operations were represented in this fashion—the C functions being named according to mnemonic descriptions of the operations. The combination of C source code, the commercial C compiler, the object code archiver and libraries, and the host executable code (which generated the microcode files) composed the *microcode assembly* subsystem of the HISSAP software development environment.

Each function generated the required number of microwords and, using the parameters sent by the calling program, modified the bit fields necessary for the execution of the HISSAP primitive. If the C source code specified that microcode instructions be generated for multiple processors, then the microcode files for those processors were created during the execution of that single host program. In this way, code could potentially be generated for all 20 processors "in parallel," thus saving the time spent using multiple source and executable files.

In addition to binary microcode, these C functions created related *user comment*, *pipeline trace*, and *error message* files. User comments that describe program event flow

were created by the programmer in conjunction with the C functions that specify microcode instructions. These comments were intended to aid in debugging program execution. Pipeline trace comments provided bit-field by bit-field descriptions of microinstruction execution. Error messages provided information about erroneous bit-field assignments or contentions (these are described in detail below).

The object code representing these functions was placed in *primitive libraries* (by using a commercial library archiving program) and served as the microcode database in the HISSAP software development environment. Medium-level programs that called the basic functions became functions themselves, thereby embodying the next level of the code hierarchy, and so on. Libraries that contained functions from these different levels were created as the HISSAP programming effort evolved. A single function call, therefore, could be responsible for generating microcode to perform as simple an operation as a sequencer jump or a more complex one such as an inner product calculation. A more detailed discussion of object code libraries is given in the section on applications-level programming.

The C-based microcode assembly "subsystem" proved to be the main tool for HISSAP primitives development (and, as will be discussed later, for the development of large-scale applications). The convenience of mnemonic-type function calls and the automation of large-volume code generation eliminated the need for "hand-coding" of primitives, while the generation of trace and error message files aided in debugging.

Example of Operation. In each HISSAP microinstruction, all aspects of hardware operation (enabled or disabled) had to be specified: memory location, direction of data flow, address generation, floating-point mode, etc. At the source level, specifications were passed as parametric arguments to the C functions. For example, a command to move (MV) a floating-point value from data memory (DM) to an arithmetic register (RF) would be of the form:

MV_DM_RF (mem_location, reg_location),

where the "locations" were represented either by hex values or by previously defined labels. The function, when called during execution of the microcode generating program, generated the correctly configured binary microcode image in a file ready for download to HISSAP by the host personal computer (PC).

C Functions as Node-Level Source Code. An important distinction should be made between the use of the term "high-level language," in the context of this report, and its use in other parallel processing literature. This report refers to high-level language (HLL) as one such as C or PASCAL that allows expedient creation of microcode without resorting to binary manipulation. In the case of HISSAP, the source code for each node was generated independently, the system architecture was usually apparent in the

function call (see previous example), and interprocessor communication was set up explicitly by the programmer (more on this later). Other references to HLL usually mean that the system architecture is not visible to the programmer, that a single piece of source code represents an entire array operation, and that a compiler exists that generates and distributes individual node microcode and handles internode communication. The programming language used by those systems might be better thought of as "high-level DISTRIBUTED language," whereas the use of C for generating HISSAP code is really programming in a "high-level NODE language."

Binary Field Editor

In the other method of generating HISSAP code, individual microwords were "hand-coded," with the use of a binary field editor. This editor allowed direct modification of individual bits within microwords and was particularly useful for creating small microcode files or for making minor changes in a large binary file. Because hardware resource assignment, counter/timer values, and sequencer instructions could be quickly manipulated, this method was also useful for debugging software and hardware.

One of the most notable features of this utility was the ability to view and update the bit fields at the bit or field level, with on-screen display of the binary, hexadecimal, and decimal values. Textual information was also shown that described the effect of the field's current value on the particular hardware subsystem controlled by that field. This information provided a user-friendly method of configuring bit fields by using multiple references of hardware functionality. Figures 2a and 2b show two examples of this utility.

A primitive-level programmer, unlike an applications programmer, needs access to microcode at the bit-field level. In a programming environment where low-level (primitive) coding is performed, a binary field editor is useful for quick manipulation of microcode. The binary editor included in the HISSAP software development environment proved to be a useful resource, especially during code debugging.

Figure 3 provides a summary depiction of the HISSAP microcode generation system.

```
12 100000 0 00 00 0 00 0000 0000 1111 1110 000 000 0 0 0 0 11111 00000 00000
   00000 00000 00000 00000 1 1 1 1 0 0 0000 0 0 0 0 1111 0 1111 1000 1000 1000
   10000 00 00 1 00 00 00 0000 00000000000000000000000000000000 0000 0 0000 1110
13 100000 0 00 00 0 00 0000 0000 1111 1111 000 000 0 0 0 0 10000 00000 00000
   00000 00000 00000 00000 1 1 1 1 0 0 0000 0 0 0 0 1111 0 1111 1000 1000 1000
   10000 00 00 1 00 00 00 0000 00000000000000000000000000000000 0000 0 0000 1110
14 100000 0 00 00 0 00 0000 0000 1111 1111 000 000 0 0 0 0 10000 00000 00000
   00000 00000 00000 00000 1 1 1 1 0 0 0000 0 0 0 0 1111 0 1111 1010 1000 1000
   10110 00 00 0 00 10 01 0000 00000000000000000000000000110001 0000 0 0000 1110
15 100000 0 00 00 0 00 0000 0000 1111 0111 000 000 0 0 0 0 11111 00000 00000
   00000 00000 00000 00000 1 1 1 1 0 0 0000 0 0 0 0 1111 0 1111 1000 1000 1000
   10000 00 00 1 00 00 00 0000 00000000000000000000000000000000 0000 0 0000 1110
16 100000 0 00 00 0 00 0000 0000 1111 1111 000 000 0 0 0 0 10000 00000 00000
   00000 00000 00000 00000 1 1 1 1 0 0 0000 0 0 0 0 1111 0 1111 1000 1000 1000
   10000 00 00 1 00 00 00 0000 00000000000000000000000000000000 0000 0 0000 1110
17 100000 0 00 00 0 00 0000 0000 1111 1111 000 000 0 0 0 0 10000 00000 00000
   00000 00000 00000 00000 1 1 1 1 0 0 0000 0 0 0 0 1111 0 1111 1000 1000 1000
   10010 00 00 0 00 10 01 0000 00000000000000000000000000110010 0000 0 0000 1110
18 100000 0 00 00 0 00 0000 0000 1111 1111 000 000 0 0 0 0 10000 00000 00000
   00000 00000 00000 00000 1 1 1 1 0 0 0000 0 0 0 0 1111 0 1111 1000 1000 1000
   10000 00 00 1 00 00 00 0000 00000000000000000000000011000 0111 0 0000 0011
──────────────────────── ADD.SAX ────────────────────────
                                                0..18..18  bit: 0
```

Figure 2a. Binary editor displaying Microcode.

```
12 100000 0 00 00 0 00 0000 0000 1111 1110 000 000 0 0 0 0 11111 00000 00000
   00000 00000 00000 00000 1 1 1 1 0 0 0000 0 0 0 0 1111 0 1111 1000 1000 1000
   10000 00 00 1 00 00 00 0000 00000000000000000000000000000000 0000 0 0000 1110
13 100000 0 00 00 0 00 0000 0000 1111 1111 000 000 0 0 0 0 10000 00000 00000
   00000 00000 00000 00000 1 1 1 1 0 0 0000 0 0 0 0 1111 0 1111 1000 1000 1000
   10000 00 00 1 00 00 00 0000 00000000000000000000000000000000 0000 0 0000 1110
14 100000 0 00 00 0 00 0000 0000 1111 1111 000 000 0 0 0 0 10000 00000 00000
   00000 00000 00000 00000 1 1 1 1 0 0 0000 0 0 0 0 1111 0 1111 1010 1000 1000
 Instruction Address: 018
   ● ● ● 28 ● ● ● 24 ● ● ● 20 ● ● ● 16 ● ● ● 12 ● ● ● 8 ● ● ● 4 ● ● ● 0
   ┌──────┬──────┬──────┬──────┬──────┬──────┬──────┬──────┐
   │  0   │  0   │  0   │  3   │  0   │  e   │  0   │  3   │
   │0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 1 0 0 0 0 0 0 0 1 1│
   └──────┴──┬───┴──────────┬──┴──────┬──┴──────┬──┴──────┘
             │  0   │  018  │  7   │  00  │  3   │
             └──────┴──────┴──────┴──────┴──────┘
 Field   Sequencer Instruction Select
 Descr     Conditional Jump                    Addr = (PLR)
18 100000 0 00 00 0 00 0000 0000 1111 1111 000 000 0 0 0 0 10000 00000 00000
   00000 00000 00000 00000 1 1 1 1 0 0 0000 0 0 0 0 1111 0 1111 1000 1000 1000
   10000 00 00 1 00 00 00 0000 00000000000000000000000011000 0111 0 0000 0011
──────────────────────── ADD.SAX ────────────────────────
                                                0..18..18  bit: 0
 Write   Jump   Toggle cursor   Help   Quit   HOME   END   CTRL+HOME   CTRL+END
```

Figure 2b. Binary editor in Microcode Edit mode.

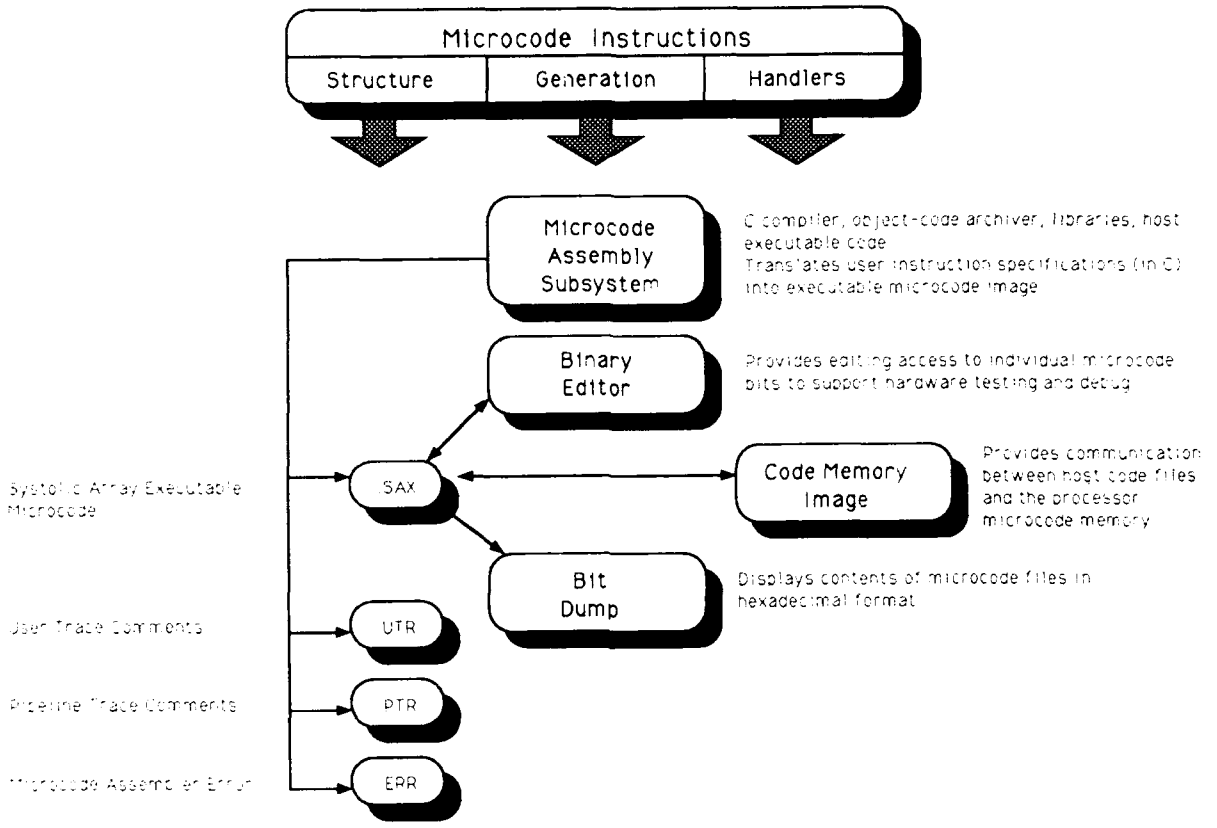


Figure 3. HISSAP Microcode Generation System.

RESOURCE ALLOCATION AND MANAGEMENT

HISSAP's 176-bit (or 64-bit) microwords required error-checking utilities in the primitive programming environment. This need was envisioned during the development of tools for HISSAP. Such tools, whether included in the final HISSAP software development environment or not, are described here.

Bit-Field Error-Checking

The bit fields defining hardware states were restricted to certain values or ranges of values. When supplying parameters to the microcode generating functions, the programmer had to be aware of the limitations on those values. However, an automated method of detecting invalid bit-field assignments was developed. Such an error-detection method benefited the HFDF software development effort. If an invalid microcode bit assignment were made, the generating function flagged the programmer with information regarding the microword location, bit field, and erroneous bit pattern.

For example, the largest immediate memory location addressable in a HISSAP microword was decimal 4095 (hex FFF). If the programmer attempted to pass a value larger than FFF as an immediate address parameter, the generating routine would detect it and produce an error message with the pertinent information.

A notable feature of this resource was its ability to check for the validity of bit fields with regard to their interaction with other related bit fields. Fields whose *individual* configurations were correct may have collectively represented a *concurrent* hardware operation that was undefined. The HISSAP code generation functions performed checks of related bit fields.

One suggested improvement of this feature would be to allow "real-time" corrections of invalid bit-field assignments during code generation. On HISSAP, once the errors were mapped, the programmer returned to the source code level, made the corrections, and recompiled. Using the proposed method, once an error was detected, the programmer would be offered the choice of changing it interactively or correcting it "off-line." This feature represents a marked increase in complexity of the code-generating support routines (not available in the off-the-shelf C compiler) and hence was not within the original scope of HISSAP code development. However, such a feature is envisioned to reduce microcode development time on future operational systems.

Bit-Field Contention Detection

Another type of microcode-generating error occurred when attempting to overwrite bit fields already defined within a microinstruction. This was frequently encountered during *software pipelining*. The hardware and the microword format were designed such that two or more operations could reside within a single microword (or sequence of microwords), provided that those operations did not require use of the same bit fields. Thus, microcode generating functions were written to configure bit fields in microwords previously created by other functions, in order to populate those microwords with multiple operations. This pipelining capability provided computational speed and efficiency but required a very rigorous programming effort.

When instructions are pipelined, careful attention must be paid to the bit fields being used in order to prevent contentions between coexisting operations. This requirement necessitates another error-checking capability within the microcode generating software. A processing scheme (especially one that uses instruction pipelining) must incorporate in its development environment a means to detect bit contentions caused by two or more operations sharing bit fields within a microword.

Typical applications programmed on HISSAP required 20 files (one for each of 16 arithmetic and 4 I/O processors) with hundreds or thousands of microwords each.

Attempts to pipeline instructions for speed relied on a system of checking bit patterns prior to their modification to see if an overwrite of a previous assignment was to occur. The programming utilities would flag the programmer when such contentions occurred during code generation. The errors were logged by microword and bit-field locations and thus provided a means for the programmer to find and fix the contentions. This mechanism was similar to those available in commercial microcode programming environments.

Resource Contention Detection

Frequently, when pipelining a sequence of instructions (particularly arithmetic operations), a data memory or operand register location was used more than once within that sequence. Occasionally, such a resource was inadvertently designated as an arithmetic source or destination register for an operation before the data currently occupying it was no longer needed. If a contention took place, then the operation requiring that previous value might have produced an erroneous result. The programming tools required the ability, during code generation, to mark operand registers and data memory locations as "in use," thereby protecting their contents from destruction by other operations, until those contents were no longer needed.

Built into the microcode generating tools for HISSAP was such a capability that, depending upon the length of the operation (and thus the number of instructions for which the register contents had to remain intact), protected operands from being overwritten. The HISSAP mechanism only alerted the programmer to an impending overwrite; a more complex tool could have redirected the operands of the infringing instruction to another set of registers.

Interprocessor Data Communication

All of the advanced algorithms implemented on HISSAP included interprocessor data communication in flow conventions defined by the mapping strategies. It proved to be the operation most sensitive to incorrect coding. Interprocessor communication was completely determined by the algorithm design; the number of data elements to be transferred at a particular time was algorithm-dependent. It was completely synchronous, in the sense that processors had to be in lock-step mode during the transfer interval. And there was no queuing system; the coding determined the time at which messages would be transferred. These conditions required precision in programming communication routines.

HISSAP had the capability of transferring data between processors at one element per clock period when in pipeline mode. Each element was transferred as a 32-bit word on a parallel bus. Data transfer occurred on a word-by-word basis, not on a packet basis. A short sequence of instructions was required to transfer a single word;

this sequence had to be executed once for each word transferred. However, since no information was contained within a block of data to indicate the transfer size, the programmer was responsible for providing the correct number of instruction sequences for each block transfer. In addition, the instructions for performing the handshaking and transfers had to contain other pertinent information at the source level, including data flow direction (port assignment) and the memory locations accessed.

Of course, the data had to be valid on the communication link during the interval that the receiver interrogated its incoming port. The programmer was responsible for sequencing the respective write and read instructions so that these timing-critical events occurred on the same clock cycle. This was done at the source code level—there was no compiler to ensure that those events lined up in time. The programmer usually initiated a transfer with a REQUEST/ACKNOWLEDGE handshake pair to synchronize the processors, followed by carefully aligned sequences of transfer instructions. It is easy to see that generating individual transfer routines for a large algorithm partitioned among 20 processors could be very time-consuming. Debugging the HFDF code when transfer errors occurred was, at best, quite tedious.

Whatever the method of message transfer (including queued, packeted messages), there must be a means of checking the integrity of interprocessor communication. In the case of HISSAP, even a single clock interval of misalignment between the interacting processors would cause failure in the transfer of data. The HISSAP utilities would have benefited by having the capability to check integrity of handshaking, and proper alignment of data transfer instructions, in the microcode of two or more interacting processors.

Such a utility would be able to recognize when a transfer was to occur at a specific point in time during the algorithm execution. Since the transfer would be a shared event, this information should be supplied in *each* of the active processors' source code files. During microcode generation, a "moderating" function, similar to one used for bit-field contention, would detect transfers and would verify that the handshake signals occurred correctly and that subsequent transfer instructions lined up properly in time. If potential errors were detected, then information regarding their location(s) in the source code would be supplied to the programmer.

The absence of such a utility in the HISSAP programming environment resulted in significant time spent debugging unsuccessful data transfer attempts. In addition, *hardware-oriented* problems in transferring data often could not be identified until correctly configured microcode was available to test the transfer operations. Uncertainty in both hardware and software integrity created difficulties in troubleshooting those operations. Therefore, based upon our experience, it is important to have a way to verify correct use of communication instructions by interacting processors.

Synchronizing Algorithm Modules

In order to coordinate the tasks performed by the processors in a parallel system, it is often useful to divide the algorithm into computational "phases." Each phase is characterized by the processors performing a number of computations; these computations depend on the results of previous phases from other processors. The processors do not necessarily need to be synchronized *during* the computations, but they DO need synchronization *between* phases, particularly if data are to be passed between processors during those interim periods. The algorithm must include a mechanism for synchronizing the processors at these times.

There were two ways in which code modules that represented phases were synchronized on HISSAP. The first of these involved local control, whereby the active processors *remained* in synchronization between the execution of adjacent routines. In developing algorithms on HISSAP, the synchronization of processors between two arbitrary modules by using this lock-step method was a critical and often painstaking task. The code blocks within EACH of the 16 processors had to contain exactly the same number of instructions (including No Operations [NOPs]) to ensure synchronization.

The other method of synchronization involved global signal control, such that each processor looped on a single instruction and sent a low level to a wired-OR circuit and waited for all processors to send this level before proceeding. This proved to be the most convenient, reliable method of interprocessor synchronization.

Either method would have benefited from a utility to check synchronization during microcode generation. Such a method would be similar to checking for data communication synchronization (as described above). A local synchronization check would count instructions designated by the programmer as "sync" instructions; a check of the global synchronization would ensure that all processors had an instruction at some "phase boundary" that performed a global signal synchronization.

The lack of such a utility, while not as critical as the data communication verification utility, nevertheless resulted in otherwise productive programming time being spent tracking down synchronization errors during HFDF software development. This was especially true when large applications began to emerge from primitive building blocks, few of which had standardized synchronization checks built in.

APPLICATIONS-LEVEL PROGRAMMING

This section addresses the issue of programming algorithms on HISSAP at the applications level. This was to be the predominant method of programming algorithms, particularly after a complete set of primitives had been made available in user libraries.

Since *applications* (or *algorithm*) programmers should not be concerned with the lowest machine-level details of instruction interaction or optimization, their programming environment must contain a suite of tools different from those provided for primitive-level programmers. The tools must provide a means to interface the primitive modules needed to form complete programs and must also provide competent run-time analysis and debugging capabilities.

This section will cover the topic of software module maintenance and interfacing, including a discussion of the need for an environment under which an applications programmer can "comfortably" create programs from primitive modules. The next section will be devoted to the subject of program analysis and debugging.

MODULE LIBRARY MAINTENANCE

As described previously, the majority of the HISSAP algorithm development effort relied on the programming, use, and maintenance of source-code descriptions of HISSAP functions and their associated object-code libraries. Libraries containing the object code (generated from C) for basic HISSAP machine operations and low-level primitives were created to link with object code describing higher level programs. The libraries were created and managed by using the library management resources packaged with the commercial C compiler.

Obviously, the need for comprehensive primitive module libraries is greatest for the applications programmer, who generates code by using function calls at the source level and who is largely isolated from the low-level coding issues. Module reusability becomes critical when many primitives have been written, compiled, and placed into such libraries.

Module Reusability

Many primitive modules were programmed by using C subroutines for implementing algorithms on HISSAP. Some modules were "algorithm-specific" and could be used only in a narrow set of applications. A module to perform signal conditioning, for example, could be employed in beamforming, direction-finding, spectral analysis, etc. However, the implementation of the signal-conditioning module would depend on the signal set and application involved. Modules of this kind had to be custom-coded to fit the desired scenario. The modules were usually created by using the lower-level primitive functions described in the previous section, with the resulting object code stored in algorithm-specific libraries.

Other modules were created as "general-purpose" utilities for use in many of the possible HISSAP processing applications. Inner products and other matrix operations,

whose complexities varied only with problem size, were typical examples. Block data movement routines were also seen as useful additions. Modules that receive such broad usage are fundamental tools to the algorithm programmer. Programming time is saved when these modules exist as "prepackaged" source code and/or object code. With this as a motivation, libraries of *reusable* modules were created during the HISSAP algorithm development effort. Libraries of these building block tools were compiled in order to prepare for the expeditious implementation of larger algorithms.

A large inventory of general-purpose subroutines is beneficial to the development of complex processing applications. Of course, it is impossible to populate such a library with routines covering all conceivable processing tasks and processor topologies. A point of diminishing returns is reached as the collection becomes unmanageable. Instead, existing routines must be reconfigurable to some degree to suit a variety of applications. The inner product module, as a hypothetical example, would reside in a library with the required number of calculations controlled by the assignment of the vector sizes. Following the parameter syntax in the C source code, the programmer would supply the vector sizes and starting memory locations. Microcode would then be generated with the correct number of computational iterations and correct data memory accesses. Such "reusability" was an intended feature of the software integration procedure on HISSAP.

However, reusability did not mature to a useful level (with the exception perhaps of the lowest level of primitive modules). Programming a mixture of reusable and algorithm-specific modules to realize application-level routines required meticulous attention to inter- and intraprocessor details. The programmer required intimate knowledge of both the existing and target configurations of the modules, as well as the program segments with which they would interact. Oftentimes, simply supplying parameters to the microcode generating functions did not correctly configure the modules; modification of the underlying source code was required, leading to multiple "versions" of a once "reusable" routine. The extent to which such modules needed modification often required significant programming time; this ultimately hindered the overall development effort. This was particularly true when the modules involved had been created by different programmers.

It must be understood that the HFDF software development on HISSAP was an evolving process. The MUSIC algorithm incorporated modules in the category of reusable, standard building blocks. They included data transfer routines, standard matrix operations, etc. Because the preliminary focus of the HFDF effort was to produce a systolic mapping of MUSIC, many of these otherwise general-purpose modules were specialized to work in accordance with that application. The initial effort did not include the standardization of code interfaces. Therefore, extra effort was required in

configuring those routines to work in applications other than MUSIC (such as the digital filter).

It warrants mentioning, as a "lesson learned," that modules targeted for a multi-application programming environment should adhere to some reusability standard. With the inclusion of a parallel compiler in future software development environments, such standardization would be built-in; however, the lack of standardization in the early HFDF application programming on HISSAP emphasizes the need for code reusability.

Primitive and Module Naming Conventions

As a footnote to the above discussion, primitives and higher level code modules should be organized according to efficient but thorough *naming conventions*. This falls under the heading of "module library maintenance" (rather than under "primitive programming environment") because the ability to program efficiently at the application level is related to well-documented code routines available to the programmer.

Not only must modules be logically named according to their major functions, but also to specific details describing each module. Real or complex data formats, matrix size limitations, and the like, can be part of the naming. Each primitive/module should carry with it, at the source and documentation level, a description of the operation in detail, input/output data requirements as needed, execution time in clock cycles, etc. Application notes would be included when appropriate.

For the majority of the HISSAP HFDF effort, code modules were named and documented appropriately at the source level. However, the project suffered from a lack of a "programmer's reference" to most of the modules. Such a reference would be, of course, a necessity in a future parallel processing software development environment.

DATA FORM COMPATIBILITY AND TRANSITION ROUTINES

Since data communication on HISSAP did not feature packetization, a significant portion of programming involved the transition of *data distributions* between algorithm stages. Most of the time, custom code was generated to perform the data configuring. This was again due to the lack of a comprehensive set of reusable modules, specifically those that would be responsible for managing the data transition details.

Consider a data distribution, perhaps output by some arbitrary algorithm stage, that is targeted for processing by another stage. Regardless of the physical source of the input data, there must be appropriate conversion of this distribution into a form consistent with that expected by the pending routine. "Form," in the context of this report, refers to (but is not limited to) the *rate* (e.g., number of input elements required per output element) and the *structure* (e.g., data in row vs. column orientation). The *sense*

in which data are stored (and accessed) in memory is dependent upon the form of its distribution, as is the *portion* of data accessed for a given computation, etc. The programming of the algorithm must track the appropriate distributions.

In general, the programmer must know the forms of the input and output data distributions used by the algorithms. This knowledge determines the programming steps required to perform any necessary form conversions. Form conversions may be as simple as accessing a sequential data set from a portion of memory and partitioning the elements into another portion of memory according to some predefined structure. An example is a matrix transpose. The source distribution is stored sequentially by row. The new structure is a matrix whose elements are stored in memory sequentially by columns, so form conversion will involve systematic "shuffling" of the source data (e.g., by offset or modulus addressing) in preparation for input to the next routine.

One example stood out in the HISSAP HFDF application. A routine to convert the form of the filter output data distribution to the form of the MUSIC input data was required. This conversion took place as a final "step" in the filtering process and used specially written code. A more efficient approach would have been to use a general-purpose module that, using parameters supplied to it at the source level, converted the filter output data distribution into the form required by the MUSIC routine. Such intermediate code routines would be easily configurable so as to avoid the stockpiling of individual routines for all possible data conversion situations in an application.

Our experience has shown that, in many cases, the creation of separate, specialized conversion routines was cumbersome and required significant programming time. It was not sufficient that the HISSAP programmer knew the data distributions when preparing transfer or conversion routines. In addition, the programmer needed intimate knowledge of the process by which the system communicated data between processors and/or input-output elements. The programming of data transfers on HISSAP required setting up synchronization and timing between processors (either locally or globally), a priori knowledge of source and destination registers, and the length (in clock cycles) of each transfer interval. This was in addition to knowledge of data flow directions, which specified the not-so-obvious port drivers and crossbar switches that required access via microcode control bits. Inattention to any of these details, even a single control bit, often caused complete system failure.

Managing many such routines to preserve cohesiveness would require significant time defining, programming, and debugging the routines. This would especially be true when all possibilities of processor connections and transfer specifications were considered. It would have been advantageous, from our experience, to maintain a library of several general-purpose transfer (conversion) routines, each reconfigurable via parameters supplied during compilation or code generation. The details of timing,

synchronization, and execution time would be transparent to the general programmer, to whom such details bear little importance during the algorithm development stage.

MODULAR PROGRAMMING ENVIRONMENT

Finally, consider the need for computer-aided software engineering (CASE) tools for software development. These tools should allow software modules to be easily interfaced during algorithm development, preferably at some graphical (flowgraph or node) representation. This clearly would be a programming level targeted for system-level programmers who have little or no knowledge of the underlying processor architecture; no such sophistication was required for the HISSAP effort.

The ability to program at a graphical level would resemble the way a processing application is designed at the system level. Application-level programmers would be able to design and code algorithms as if they were drawing signal-flow graphs. Block diagrams drawn in the graphical environment would represent the highest level of coding; each block would correspond to a function residing in an application library. Collections of these blocks would comprise a black-box level of programming.

Each individual block would contain a functional language description of the operation, much like the C functions written for HISSAP operations. The parameters required for each block (function) would be supplied by the programmer who would use a text editor or menu-driven system. A system of cross-checking the validity of parameter assignments, both within the block and with reference to "neighboring" blocks, would be performed by the CASE tools. Data form compatibility, as described above, would be confirmed (or automatically configured) as well.

Many commercial CASE programming environments are available for a variety of processing systems, including parallel systems and DSP microprocessor systems. An advanced discussion of these and related tools (including parallelizing compilers) is given in reference 6.

PROGRAM ANALYSIS AND DEBUGGING UTILITIES

In any programming environment, it is useful to have a method of debugging the software under development. In a parallel programming environment, where multiple processors handling different data and performing different computations must be coordinated, this is essential.

One need not search long to find excellent commercial packages, designed for popular uniprocessors, that aid programmers in debugging their code. Most noteworthy

in our experience is the CodeView* debugging utility (packaged with many Microsoft* programming languages) designed for use in developing code for the Intel 80×86 family of microprocessors. CodeView features several options for debugging a compiled program, including full-speed operation with or without breakpoints, single-step operation with the ability to trace into external subroutines, data memory and processor register watch points with the ability to alter their contents, and processor stack and status register observation.

The collection of utilities used for downloading and executing microcode, viewing and modifying data in HISSAP memory, and performing hardware diagnostics composed the custom HISSAP operating system named "SAPMASTER." This operating system was implemented on the host PC, written in C and host assembly languages. A block-level diagram of this operating system is shown in figure 4.

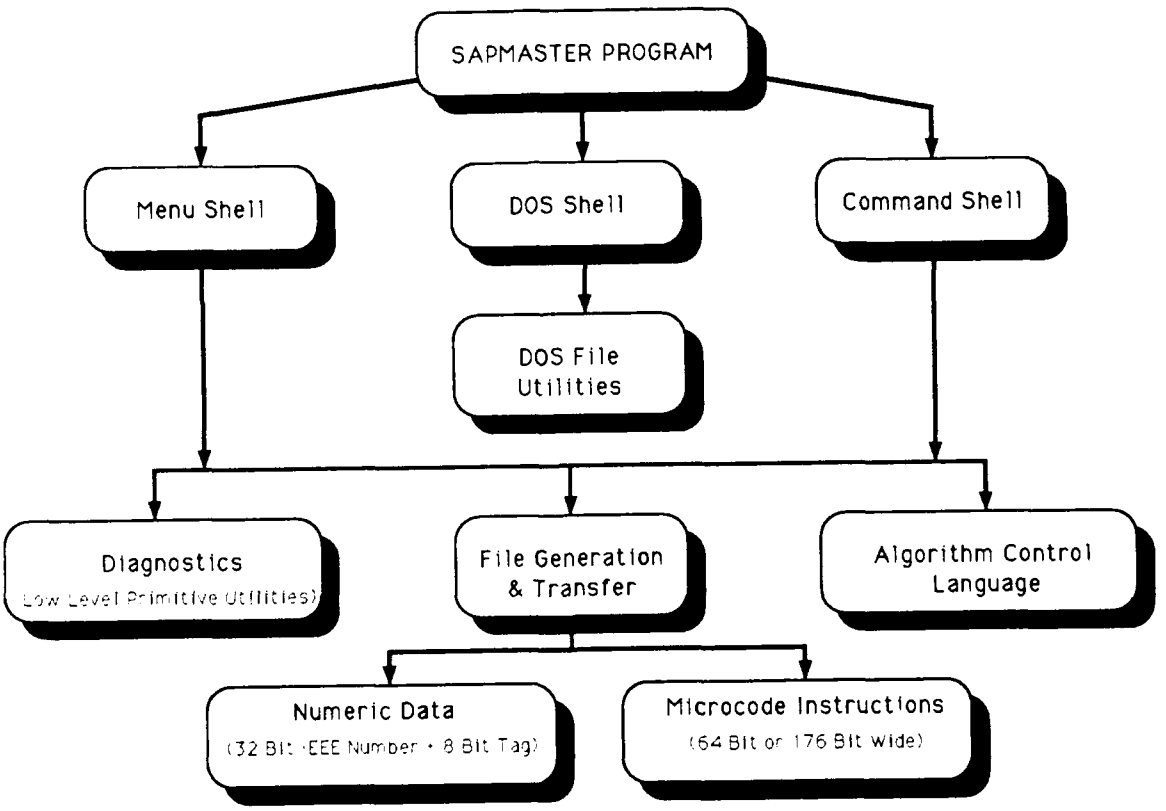


Figure 4. HISSAP Operating System, "SAPMASTER."

The intent of this operating system was threefold:

*CodeView and Microsoft are trademarks of Microsoft Corporation.

1. Provide a software development environment for the creation/analysis of data files input to or output from HISSAP algorithm mappings;

2. Provide applications programmers with access to the diagnostic utilities resident in the HISSAP hardware for isolating faults in the execution of microcode; and

3. Provide an ability to execute algorithms in "mixed mode," i.e., with computations shared between host and HISSAP.

Many of SAPMASTER's features were chosen based on the CodeView package referenced above and proved indispensable in the development of the HFDF system software. Features included a single-step execution mode, a program trace utility, a multiple-window utility for viewing the data memory contents of multiple processors simultaneously (updated as desired), data and code memory content interrogation utilities, facilities for modifying data and code memory locations, and a binary field editor for modifying microcode files. There were other features, but the ones mentioned above were the most important from a development standpoint and will be described in more detail.

Constraints on time and budget, along with the physical characteristics of the HISSAP hardware, combined to prevent what could be considered a "complete" software debugging environment. Specifically, features missing or inoperable during development included breakpoint/restart capabilities, and interrogation of floating-point operand registers. The absence of these features from the HISSAP environment at times caused difficulty in the HFDF software development. The function and importance of these "missing" features, along with those present in the system, are discussed below. Suggested enhancements to the existing features are also discussed.

PROGRAM EXECUTION UTILITIES

In debugging programs on HISSAP, several methods were available for executing all (or portions) of a program at varying clock speeds including single-step. The user selected which clock frequency and the number of cycles to be run. These utilities (especially single-step operation) were valuable in the algorithm development phase and were most powerful when used in conjunction with the utilities used for viewing memory contents and the program sequence.

For instance, when stepping through a portion of code, the user could observe the modification of data in active regions of memory as scheduled computations or data movement were executed. Correct operation of the code was determined by comparing these data to predetermined results. Portions of code known to operate correctly could

be executed at full speed ("skipped over" in a sense) until the code in question was reached. The code in question could then be scrutinized under slower execution.

Figure 5 shows a depiction of the screen used to view selected memory contents ("watches") in various processors. All 16 arithmetic processors and 4 input/output processors are shown enabled, although the user could specify fewer processors (and more data watches per processor) if desired.

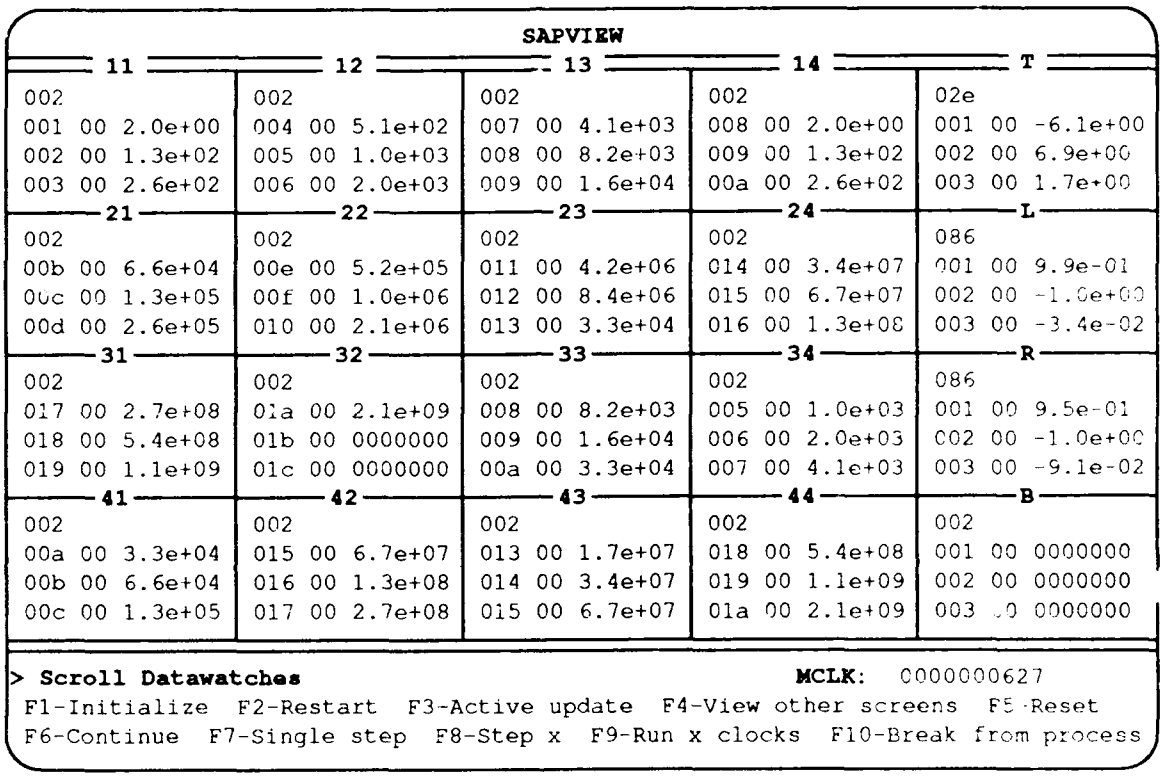


Figure 5. SAPVIEW Program Execution and Data Watch Utility.

PROGRAM TRACE AND DATA SNAPSHOT UTILITIES

In order to keep a "record" of the program execution, there were two utilities: *program trace* file creation and *data snapshot* file creation. Both file types were created during program execution by the HISSAP operating environment.

The program trace history utility made a record of the 4096 most recent program counter values for all 20 processors, thus keeping a record of the instruction sequence. Along with the program counter values, there were the hardware branch patterns responsible for the execution of the subsequent instructions. This information was most

useful when a processor's instruction sequence depended upon handshaking with a neighboring processor. Our experience with this utility was significant, in that many times improper execution could be located, and its cause determined, by the trace files.

Figure 6 shows an example of the contents of one processor's trace history. Included in the history are the time-ordered program counter (PC) values, hardware branch conditions (flags), and pipeline trace comments associated with each instruction executed during the run.

```
DELAY  PC   FLAG        PIPELINE COMMENTS

0019  0002 0   Wait for host to setup JMAP address and assert START flag
0018   0003  0 Jump to address in JMAP register.
0017   02f0  0 Beginning of module, address: (hex) 2f0
0016   02f1  0
0015   02f2  0
0014   02f3  0
0013   02f4  0 Perform multiply.  Place result in register file: L01.
0012   02f5  0
0011   02f6  0
0010   02f7  0
000f   02f8  0
000e   02f9  0
000d   02fa  0
000c   02fb  0
000b   02fc  0 Move result into data memory for next multiply.
000a   02fd  0
0009   02fe  0
0008   02ff  0 Multiply completed.  Wait for host to reset START flag.
0007   02ff  0 Multiply completed.  Wait for host to reset START flag.
0006   0300  0 End of module.  Return to address: 0.
DELAY = 006   (hex)                        Processor = 11
```

Figure 6. Program Trace History Utility.

The data snapshot utility worked similarly but was more powerful in that the user specified which data memory locations were to be interrogated, how many instructions were executed between interrogations, and which processors were involved in the records. This capability allowed a more selective viewing of data, and the execution rate between snapshots could be either full-speed or single-step. An added feature was the capability of saving snapshot states to files for off-line analysis.

The data snapshot feature generated a history of the computational results of a program, at user-selectable intermediate steps of the algorithm. The capture of such information was a crucial part of algorithm debugging. While not always used to its full potential on HISSAP, this utility did play an important part in early algorithm

development, particularly during the MUSIC implementation. The data snapshot feature is envisioned as an integral part of any parallel programming environment.

Figure 7 is a diagram of the data snapshot concept.

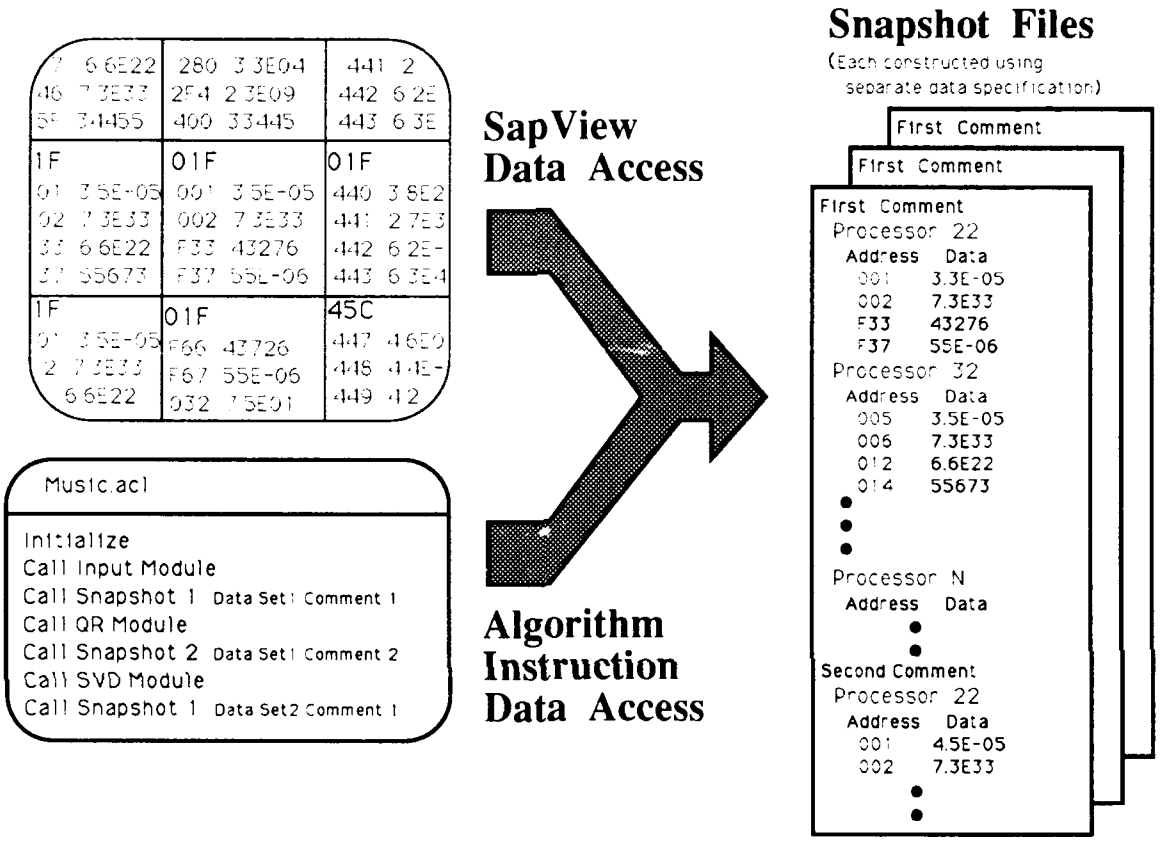


Figure 7. HISSAP Data Snapshot Utility.

BREAKPOINT UTILITIES

A lack of operational breakpoint capabilities represented a shortcoming of the HISSAP debugging utilities. While such capabilities were originally designed into the HISSAP hardware and development software, they were never fully implemented. Their absence was noticeable, since many times it was necessary to halt the processors in the middle of computations to inspect intermediate results in data memory. Throughout the HISSAP programming effort, this interruption was achieved by performing combinations of full-speed and single-step processor clocking, until the desired portion of code was reached. This process required knowledge of the differential clock count necessary to advance the sequencer to the target instruction. When the count was not known exactly, educated guessing and "hit-and-miss" clocking was a last resort.

Breakpoint capability, available in most commercial software debugging utilities, provides a user-friendly method of controlling program execution, such that processor activities can be monitored. Some of the more advanced utilities allow breakpoints to be set at both low and high code levels. It is important to provide both levels of access in order to benefit machine-level programmers as well as algorithm researchers. In the case of HISSAP, the ability to set breakpoints at both the microinstruction and C source levels would have markedly decreased the time spent debugging software. The restart capability would be required, of course, coexistent with that of multiple breakpoint setting.

OPERAND REGISTER AND STACK INTERROGATION

Another feature missing from the HISSAP programming utilities was the ability to view (and modify) the contents of arithmetic operand registers and the stack. This omission was a result of the hardware characteristics. The underlying cause was that the host diagnostic bus did not connect to the register file chips or to the hardware stack register.

For computations performed with HISSAP's floating-point units, the source operands had to reside in operand registers; similarly, results could only be written to registers. Debugging algorithms on HISSAP frequently required the interrogation of operand registers during computational phases. This need was most common when software under development produced erroneous results. The method for viewing register contents was to insert "debugging code" (which wrote the register contents to an unused portion of data memory) into the algorithm code, whereupon the memory interrogation utilities were invoked and the register contents inspected. This method was very time-consuming and required the programmer to change the source code, recompile, generate the modified microcode, download it to the processors, and execute the program until the desired operations finished.

Viewing the program stack contents directly is another indispensable resource for a parallel programming environment. When incorrect HISSAP program execution resulted after subroutine calls, or any other operations using stack manipulation, the inability to interrogate stack registers often left programmers without an indication of the error source. Considerable time was often spent tracing processor histories and tearing apart microcode to locate these sources.

ALGORITHM DESIGN AND ASSESSMENT UTILITIES

One tool necessary to shorten the algorithm development time on a parallel processing system is an *analysis* and *simulation* utility. Such a utility allows algorithms to be

tested completely under software emulation and makes available performance assessments of the simulation. Since multiprocessor implementations of the algorithm may be tested and changed quickly according to simulation output, this utility allows candidate algorithms to be almost completely designed while isolated from the target system.

PC-MATLAB AS AN ALGORITHM ANALYSIS TOOL

PC-MATLAB is linear-algebra-based analysis software package. Included in the tools provided with PC-MATLAB are many important matrix-based functions, some of which are used in the general MUSIC algorithm (e.g., the singular value decomposition and the QR decomposition). Also included is a comprehensive set of digital signal-processing routines (e.g., FFTs, filter designs and implementation, etc.).

It was stated before that algorithm analysis performed by using PC-MATLAB preceded the mapping of those algorithms onto HISSAP (the analysis of MUSIC by using PC-MATLAB was performed by Dr. S. I. Chou of NOSC*; results of his work are given in reference 7). This work was initially performed in an effort to verify the computational performance of target algorithms such as MUSIC. Later, the work was used to simulate the partitioning of the algorithms among different configurations of "multiple processors" so that the performances of candidate HISSAP implementations could be analyzed.

Since PC-MATLAB ran on a sequential processing machine, performance characteristics such as execution time and interprocessor communication could not be evaluated automatically. However, such measures could be extrapolated by using HISSAP's known operational specifications. Those results provided guidance in selecting "optimum" parallel algorithm configurations.

An added value of using PC-MATLAB as an assessment tool was that selected intermediate results of an algorithm's execution could be observed, thus giving insight into the validity of a particular implementation of that algorithm. Those simulation results were saved for eventual comparison with results output at respective locations in the HISSAP implementation of the algorithm.

Because of the rigorous effort required to program HISSAP (especially in the early stages of algorithm mapping), the simulation of algorithms on PC-MATLAB proved to be one of the most important features of the HISSAP software development environment.

---

*Private conversation with Dr. S. I. Chou, Naval Ocean Systems Center, San Diego, CA, 1 March 1991.

CONCLUSION

The HISSAP HFDF programming experience was unique from software development on a hypothetical, "in-place" parallel processing system because some of the HISSAP software development tools were implemented concurrently with the MUSIC algorithm development. As time went on, additional features and improvements to existing utilities were adopted based upon input from microcode/algorithm programmers. On a larger scale, the constructs of the microcode and operational aspects of some of the software development utilities changed as the HISSAP system hardware was modified throughout the project. These conditions indicate the extent to which the HISSAP project was an important learning experience in the design and implementation of software development environments for parallel programming.

This report has attempted to describe the lessons learned from the HISSAP HFDF software development experience within the specific scope of that project's objectives. Again, software development environments for future Navy systems will undoubtedly feature parallelizing compilers, algorithm design and performance profilers, graphical user interfaces, and the like. If any of the lessons learned from the HISSAP effort could be directly extended to general parallel processing systems, they would be related to those advanced features.

REFERENCES

1. Symanski, J. J. 1983. "Implementation of Matrix Operations on the Two-Dimensional Systolic Array Testbed," *Proc. SPIE Technical Symposium*, 21–26 August. San Diego, CA.

2. Drake, B. L., F. T. Luk, J. M. Speiser, and J. J. Symanski. 1987. "SLAPP: A Systolic Linear Algebra Parallel Processor," *IEEE Computer* (July), vol. 20, no. 7, pp. 45–49.

3. Henderson, T. B., J. J. Symanski, and K. Bromley. 1989. "Software Development on the Video Analysis Transputer Array," *Proc. 1st Conference of the North American Transputer Users Group*, 5–6 April. Salt Lake City, UT.

4. Loughlin, J. P. 1987. "NOSC Advanced Systolic Array Processor (ASAP)," *Proc. SPIE Technical Symposium*, 18–19 August, vol. 827-13, Real Time Signal Processing X. San Diego, CA.

5. Loughlin, J. P., and F. M. Tirpak, Jr. 1990. "Systolic Signal Processor/High Frequency Direction Finding Final Test Report," NOSC TR 1369 (October). Naval Ocean Systems Center, San Diego, CA.

6. Lauwereins, R., M. Engels, J. Peperstraete, E. Steegmans, and J. Van Ginderdeuren. 1990. "GRAPE: A CASE Tool for Digital Signal Parallel Processing," *IEEE ASSP Magazine* (April), vol. 7, no. 2, pp. 32–43.

7. Loughlin, J. P. 1988. "Multiple Signal Classification (MUSIC) Hosted on High Speed Systolic Array Processor (HiSSAP)," *Proc. SPIE 32nd Annual International Technical Symposium*, vol. 977–34, San Diego, CA.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1 AGENCY USE ONLY *(Leave blank)*	2 REPORT DATE	3 REPORT TYPE AND DATES COVERED
	June 1991	Final: Mar 88 – Mar 91

4 TITLE AND SUBTITLE

SOFTWARE DEVELOPMENT ON THE HIGH-SPEED SYSTOLIC ARRAY PROCESSOR (HISSAP):
Lessons Learned

5 FUNDING NUMBERS

Navy Industrial Fund
PROJ: EE34
WU: DN308022

6 AUTHOR(S)

F. M. Tirpak, Jr.

7 PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Ocean Systems Center
Code 761
San Diego, CA 92152–5000

8 PERFORMING ORGANIZATION REPORT NUMBER

NOSC TR 1429

9 SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Office of Naval Technology
Laser and Microelectronics Block
800 North Quincy Street
Arlington, VA 22217

Naval Sea Systems Command
NAVSEA PMS–412
Washington, DC 20362

10 SPONSORING/MONITORING AGENCY REPORT NUMBER

11 SUPPLEMENTARY NOTES

12a DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b DISTRIBUTION CODE

13 ABSTRACT *(Maximum 200 words)*

This report documents the software integration procedure on the Naval Ocean System Center's High-Speed Systolic Array Processor (HISSAP) testbed. Procedures are evaluated as they apply to algorithm development on future Navy signal-processing systems.

14 SUBJECT TERMS

systolic array technology
matrix processor
algorithm programming

15 NUMBER OF PAGES

36

16 PRICE CODE

17 SECURITY CLASSIFICATION OF REPORT	18 SECURITY CLASSIFICATION OF THIS PAGE	19 SECURITY CLASSIFICATION OF ABSTRACT	20 LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAME AS REPORT

21a NAME OF RESPONSIBLE INDIVIDUAL	21b TELEPHONE *(include Area Code)*	21c OFFICE SYMBOL
F. M. Tirpak, Jr.	(619) 553–2526	Code 761

INITIAL DISTRIBUTION

Code 0012	Patent Counsel	(1)
Code 0144	R. November	(1)
Code 76	F. M. Tirpak	(1)
Code 761	Dr. G. W. Byram	(5)
Code 761	F. M. Tirpak, Jr.	(5)
Code 952	J. Puleo	(1)
Code 961	Archive/Stock	(6)
Code 964B	Library	(3)

Defense Technical Information Center
Alexandria, VA 22304-6145 (4)

NOSC Liaison Office
Washington, DC 20363-5100 (1)

Center for Naval Analyses
Alexandria, VA 22302-0268 (1)

Naval Air Development Command
Warminster, PA 18974-5000 (2)